J. of Mult.-Valued Logic & Soft Computing., Vol. 11, pp. 437–465
Reprints available directly from the publisher
Photocopying permitted by license only

Exact and Heuristic Minimization of the Average Path Length in Decision Diagrams[1]

SHINOBU NAGAYAMA[*], ALAN MISHCHENKO[†],
TSUTOMU SASAO[*,‡] AND JON T. BUTLER[§]

[*] *Department of CSE, Kyushu Institute of Technology, Iizuka 820-8502, Japan.*
[†] *Department of EECS, UC Berkeley, Berkeley CA 94720, USA.*
[‡] *Center for Microelectronic Systems, Kyushu Institute of Technology, Iizuka 820-8502, Japan.*
[§] *Department of ECE, Naval Postgraduate School, Monterey, CA 93943-5121, USA.*

In a decision diagram, the average path length (APL) is the average number of nodes on a path from the root node to a terminal node over all assignments of values to variables. Smaller APL values result in faster evaluation of the function represented by a decision diagram. For some functions, the APL depends strongly on the variable order. In this paper, we propose an exact and a heuristic algorithm to determine the variable order that minimizes the APL. Our exact algorithm uses branch-and-bound. Our heuristic algorithm uses dynamic reordering, where selected pairs of variables are swapped. This paper also proposes an exact and a heuristic algorithm to determine the pairs of binary variables that reduce the APL of multi-valued decision diagrams (MDDs) for a 4-valued input 2-valued output function. Experimental results show that the heuristic algorithm is much faster than the exact one but produces comparable APLs. Both algorithms yield an improvement over an existing algorithm in both APL and runtime. Experimental results for 2-valued cases and 4-valued cases are shown.

Keywords: BDD, MDD, average path length (APL), node traversing probability, edge traversing probability, branch-and-bound, sifting algorithm.

1 INTRODUCTION

Binary decision diagrams (BDDs) [5] and multi-valued decision diagrams (MDDs) [15] are extensively used in logic synthesis [10], logic simulation [1, 13, 17], software synthesis [2, 14], and pass transistor logic (PTL) [3, 29, 30]. These applications use decision diagrams to evaluate logic functions, and the evaluation time is proportional to the average path length (APL) in the decision diagram. Therefore, minimization of the APL leads to faster evaluation of the logic function. Particularly, in logic simulation using decision diagrams [1, 13, 17], minimization of the APL reduces the simulation time substantially because logic functions are evaluated many times with different test vectors.

Minimization of the APL can also be applied to logic synthesis. A method for functional decomposition [32] uses BDDs to detect Boolean divisors. The quality of a divisor is measured by the number of don't-cares it provides for the

[1] Since this paper is reformatted to A4 size, page numbers differ from original one.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2005		

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Exact and Heuristic Minimization of the Average Path Length in Decision Diagrams	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Postgraduate School,Department of Electrical and Computer Engineering,Monterey,CA,93943**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**21**	

minimization of the quotient. The don't-cares are generated by the paths in the BDD that lead to the terminal nodes. The shorter the paths, the more don't-care minterms they contain. Therefore, minimizing the APL in BDDs can improve the quality of decomposition.

In pass transistor logic (PTL) synthesis, the circuits are derived directly from BDDs representing logic functions. In this case, the longer paths in BDDs cause larger voltage drop and larger delay. This problem can be solved by inserting buffers in long paths [3]. Minimizing the APL in the BDD can reduce the number of buffers that must be inserted.

In this paper, we propose an exact APL minimization algorithm based on the branch-and-bound algorithm. This algorithm finds an optimum variable order much faster than exhaustive search, which enumerates all possible variable orders. However, the exact method is time-consuming for functions with many inputs. To minimize the APL of such functions in a reasonable time, we propose a heuristic algorithm based on dynamic variable reordering.

This paper is organized as follow. Section 2 contains the necessary terminology and definitions. Section 3 introduces lower bounds on the APL. Section 4 proposes an exact and a heuristic minimization algorithm for the APL. Section 5 considers the paired ordering of binary variables. Section 6 shows the efficiency of the algorithms using benchmark functions. Experimental results for 2-valued cases and 4-valued cases are shown. The Appendix includes the proofs of theorems.

2 PRELIMINARIES

We assume that the reader is familiar with the basic terminology of reduced ordered binary decision diagrams (ROBDDs) [5] and reduced ordered multi-valued decision diagrams (ROMDDs) [15]. In the following, a BDD and an MDD mean an ROBDD and an ROMDD. DD means either BDD or MDD.

Definition 2.1 *Let x be an r-valued variable, and let $c \in \{0,1,\ldots,r-1\}$. Then, $P(x=c)$ denotes the probability that x has value c.*

Definition 2.2 *In a DD, a sequence of edges and non-terminal nodes leading from the root node to a terminal node is a* **path***. The number of edges in the path is the* **path length***.*

Note that the sequence of edges in a path p_i of a DD corresponds to an assignment of values a_i to the specific variables associated with those edges in the DD. We say that such an assignment a_i *selects* path p_i. Similarly, if an assignment of values c_i to *all* variables agrees with a_i for all variables assigned in a_i, we also say c_i *selects* path p_i.

Definition 2.3 *In a DD for an n-variable function, the* **path probability** *of a path p_i, denoted by $PP(p_i)$, is the probability that the path p_i is selected in all assignments of values to the r-valued variables. $PP(p_i)$ is given by*

$$PP(p_i) = \sum_{\vec{c} \in C_i} P(x_1 = c_1) \times P(x_2 = c_2) \times \ldots \times P(x_n = c_n),$$

where C_i denotes a set of assignments of values to the variables selecting the path p_i, $\vec{c} = (c_1, c_2, \ldots, c_n)$, each $c_j \in \{0,1,\ldots,r-1\}$, and $P(x_j = c_j)$ is the probability x_j has value c_j.

Definition 2.4 *The* **average path length***, or* **APL***, in a DD is given by:*

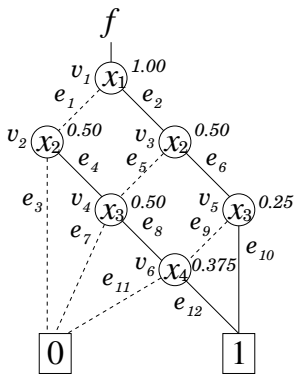
$$APL = \sum_{i=1}^{N} PP(p_i) \times l_i,$$

where i indexes the paths, N denotes the number of paths, and l_i denotes the path length of path p_i.

Definition 2.5 *The* **node traversing probability** *of a node v, denoted by $NTP(v)$, is the probability that an assignment of values to the variables selects a path that includes the node v.*

Definition 2.6 *The* **edge traversing probability** *of an edge e, denoted by $ETP(e)$, is the probability that an assignment of values to the variables selects a path that includes the edge e.*

Note that the node traversing probability of the root node in a decision diagram is 1.0, since all paths start from the root node.



(a) BDD

Path p_i	$PP(p_i)$	Path length l_i
p_1	0.25	2
p_2	0.125	3
p_3	0.0625	4
p_4	0.0625	4
p_5	0.125	3
p_6	0.0625	4
p_7	0.0625	4
p_8	0.0625	4
p_9	0.0625	4
p_{10}	0.125	3

(b) PPs and path lengths

Figure 1: Example of node traversing probability in a BDD.

Lemma 2.1 *[27] The node traversing probability of node v is the sum of the edge traversing probabilities of all incoming edges to v. Also, the node traversing probability of node v is the sum of the edge traversing probabilities of all outgoing edges from v.*

Proof. See Appendix.

From Lemma 2.1, the following relation holds:

$$ETP(e) = P(x = c) \times NTP(v),$$

where $P(x = c)$ is the probability x has a value c, v is a node representing a variable x, and e is an outgoing edge corresponding to the value c of v.

Theorem 2.1 *[27] The APL is equal to the sum of the edge traversing probabilities of all edges. Also, the APL is equal to the sum of the node traversing probabilities of all the non-terminal nodes.*

Proof. See Appendix.

From Theorem 2.1, we have the following:

$$APL = \sum_{i=1}^{N_e} ETP(e_i) = \sum_{j=1}^{N_v} NTP(v_j),$$

where N_e and N_v denote the number of edges and non-terminal nodes, respectively.

Example 2.1 *Consider the BDD in Fig. 1(a), where solid lines and dotted lines denote 1-edges and 0-edges, respectively. For simplicity, assume that* $P(x_i = 0) = P(x_i = 1) = 0.50$ $(i = 1,2,3,4)$. *This BDD has* 10 *different paths: path* p_1 *is* (v_1, e_1, v_2, e_3), *path* p_2 *is* $(v_1, e_1, v_2, e_4, v_4, e_7)$, *…, and path* p_{10} *is* $(v_1, e_2, v_3, e_5, v_5, e_{10})$. *The* $PP(p_i)$ *and path length of each path* p_i *are listed in Fig. 1(b). Therefore, by Definition 2.4,*

$$APL = \sum_{i=1}^{10} PP(p_i) \times l_i = 3.125.$$

By using node traversing probabilities, we can compute this APL as follows: First, we have $NTP(v_1) = 1.00$ *for root node* v_1. *Then,* $NTP(v_2) = ETP(e_1) = P(x_1 = 0) \times NTP(v_1) = 0.50$ *and* $NTP(v_3) = ETP(e_2) = P(x_1 = 1) \times NTP(v_1) = 0.50$. *Similarly,*

$$\begin{aligned} NTP(v_4) &= P(x_2 = 1) \times NTP(v_2) + P(x_2 = 0) \times NTP(v_3) = 0.50, \\ NTP(v_5) &= P(x_2 = 1) \times NTP(v_3) = 0.25, \quad \textit{and} \\ NTP(v_6) &= P(x_3 = 1) \times NTP(v_4) + P(x_3 = 0) \times NTP(v_5) = 0.375. \end{aligned}$$

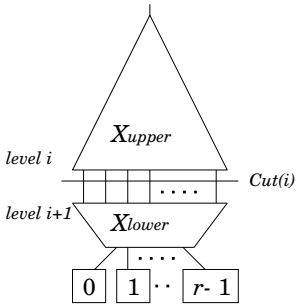


Figure 2: Partition of MDD.

Thus, we obtain

$$APL = \sum_{i=1}^{6} NTP(v_i) = 3.125.$$

Similarly, we can compute the APL using the edge traversing probabilities. *(End of Example)*

Consider a multiple-output function $F = (f_0, f_1, \ldots, f_{m-1})$: $R^n \rightarrow R^m$, where $R = \{0, 1, \ldots, r-1\}$, and n and m denote the number of input and output variables, respectively. In this paper, we use shared MDDs (SMDDs) to represent multiple-output function F. For reasons that will be clear later, we view the APL of an SMDD as the sum of the APLs of the individual MDDs for each component function f_i.

3 LOWER BOUNDS ON APL

In this section, we derive lower bounds on the APL. Such bounds are used to reduce the computation time in the algorithm, as discussed later.

Definition 3.1 *Suppose an MDD is partitioned into two parts as shown in Fig. 2. Here, X_{upper} denotes the variables above or in level i, X_{lower} denotes the variables below or in level $i+1$, and $Cut(i)$ denotes a set of edges connecting the nodes above or in level i with the nodes below or in level $i+1$.*

Note that the nodes are indexed by i starting with the root node at level 1. The nodes just below have $i = 2$, etc..

Definition 3.2 *$ETP(Cut(i))$ denotes the sum of edge traversing probabilities of edges in $Cut(i)$, and is given by*

$$ETP(Cut(i)) = \sum_{e \in Cut(i)} ETP(e).$$

Lemma 3.1 *Suppose an SMDD represents a multiple-output function $F = (f_0, f_1, \ldots, f_{m-1})$. Then,*

$$ETP(Cut(i)) = m_U,$$

where m_U is the number of the root nodes of the multiple-output function F above or in level i.

Proof. See Appendix.

Corollary 3.1 *Suppose an MDD represents a single-output function f. Then,*

$$ETP(Cut(i)) = 1.0.$$

Lemma 3.2 *Let*

$$Cut'(i) = \{e \mid e \in Cut(i), \text{ such that } e \text{ is incident to only non-terminal nodes}\}.$$

Then, for every permutation of X_{upper},

$$ETP(Cut'(i)) = c_i,$$

where $c_i \leq m_U$.

Proof. See Appendix.

Theorem 3.1 *Consider an SMDD for multiple-output function F. Let L be the sum of the node traversing probabilities of the non-terminal nodes below or in level $i+1$. Let m_L be the number of root nodes for F below or in level $i+1$. Then, for any permutation of X_{lower} and any permutation of X_{upper},*

$$ETP(Cut'(i)) + m_L \leq L.$$

Proof. See Appendix.

Theorem 3.2 *Consider an SMDD for multiple-output function F. Let U be the sum of the node traversing probabilities of the non-terminal nodes above or in level i. When the order of X_{upper} is fixed,*

$$U + ETP(Cut'(i)) + m_L \leq APL.$$

Proof. See Appendix.

Corollary 3.2 *Consider an SMDD of multiple-output function F. Let U and L be the sums of the node traversing probabilities of the non-terminal nodes above and below or in level i, respectively. Then*

$$max\{L,U\} \leq APL.$$

4 MINIMIZATION OF APL

Since the APL in a DD (BDD or MDD) depends on the variable order, the APL minimization problem can be formulated as follows:

Problem 4.1 *Given a DD for a logic function f, find a variable order that produces the minimum APL.*

4.1 Change of the APL during Swapping Two Adjacent Variables

Our APL minimization algorithms go from one variable order to another variable order by a sequence of steps that swap pairs of adjacent variables. A part of the algorithms that has a significant effect on computation time is updating the APL after swapping each pair of adjacent variables. This section describes a fast method to update the APL after the swap of two adjacent variables.

Theorem 4.1 *Let U be the sum of the node traversing probabilities of non-terminal nodes above or in level $i-1$, and let L be the sum of the node traversing probabilities of non-terminal nodes below or in level $i+2$. Then, after the variable swap of level i with level $i+1$, U and L remain unchanged.*

Proof. See Appendix.

Theorem 4.1 shows that *the previously computed node traversing probabilities need not be repeated in computing the new APL caused by the swap of two adjacent variables*. Fig. 3 illustrates a subgraph of level i and level $i+1$ in the BDD when two adjacent variables are interchanged. Since the principles of variable swap for the binary case and the multi-valued case are the same, we describe only the binary case. The details of variable swaps for the multi-valued case are discussed in [18]. A subgraph composed of BDD nodes involved in the variable swap belongs to one of the six classes shown in Fig. 3. For each class, the figure on the left occurs before the swap, while the figure on the right occurs as a result of the swap. In Fig. 3, only cases (e) and (f) do not change the APL, while other cases change the APL. For example, in case (a), the node traversing probabilities of nodes v_2 and v_3 are changed as a result of the swap. Before the swap, the node traversing probabilities of v_2 and v_3 are given by:

$$\begin{aligned} NTP(v_2) &= ETP(e_0) = P(x_i = 0) \times NTP(v_1) \\ NTP(v_3) &= ETP(e_1) = P(x_i = 1) \times NTP(v_1), \end{aligned}$$

where e_0 and e_1 denote the edges from v_1 to v_2 and from v_1 to v_3, respectively. On the other hand, after the swap, the node traversing probabilities of v_2 and v_3 are:

$$\begin{aligned} NTP(v_2) &= P(x_{i+1} = 0) \times NTP(v_1) \\ NTP(v_3) &= P(x_{i+1} = 1) \times NTP(v_1). \end{aligned}$$

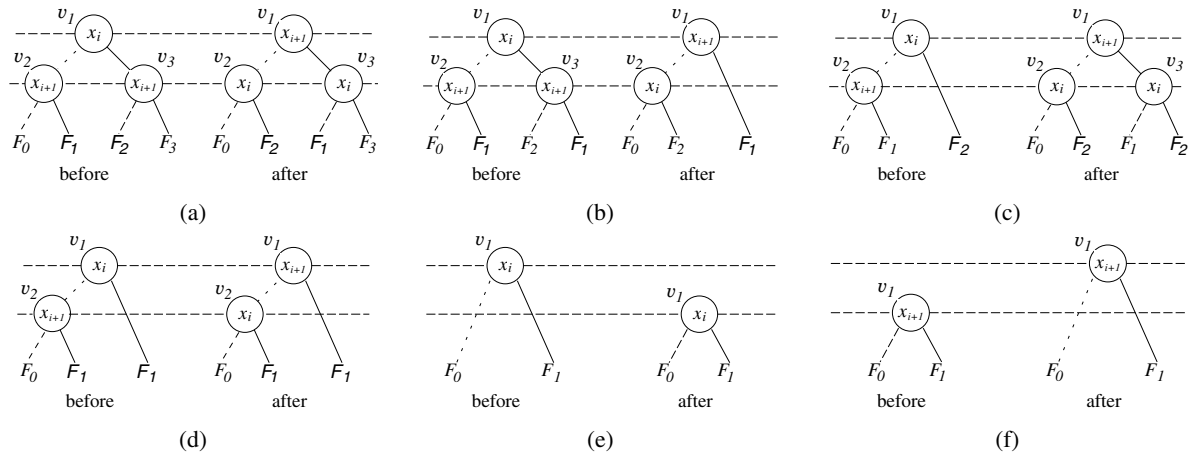


Figure 3: Six cases of exchanging two adjacent variables.

When $P(x_i = 0) = P(x_{i+1} = 0)$ and $P(x_i = 1) = P(x_{i+1} = 1)$, the node traversing probabilities of v_2 and v_3 do not change after the swap. Therefore, in case (a), the APL is changed by the edge traversing probabilities of outgoing edges from v_1. Similarly, in other cases except for (e) and (f), the APL is changed by the edge traversing probabilities of outgoing edges from the root node of a subgraph. Note that from Theorem 4.1, we consider only the edges from the root node to nodes in level $i+1$ to update the APL.

We summarize the strategy for updating the APL as follows:

1. Before the swap, for each subgraph involved in the swap, the edge traversing probabilities of the edges from the root node of a subgraph to nodes in level $i+1$ are subtracted from 1) the APL and from 2) the node traversing probabilities of nodes in level $i+1$.

2. After the swap, for each subgraph, the edge traversing probabilities of edges from the root node of a subgraph to nodes in level $i+1$ are re-calculated.

3. The calculated edge traversing probabilities are added to 1) the APL and to 2) the node traversing probabilities of nodes in level $i+1$.

Example 4.1 *Fig. 4 shows BDDs for logic function* $f = x_1x_4 \vee x_2x_4 \vee x_3$. *Fig. 4(a) shows the BDD with the variable order* (x_1,x_2,x_3,x_4), *top to bottom. For simplicity, assume that* $P(x_i = 0) = P(x_i = 1) = 0.50$ $(i = 1,2,3,4)$. *Then, the APL of the BDD in Fig. 4(a) is* 2.875. *In this BDD, we consider the swap of variables* x_2 *and* x_3. *During such a swap, case (b) applies to node* v_2 *and case (f) applies to node* v_4. *Performing the swap leads to the BDD shown in Fig. 4(b). Note that the swap decreases the APL by* 0.25 *because the node* v_4 *after the swap does not have the incoming edge from node* v_2. *The node traversing probabilities associated with nodes* v_2 *and* v_3 *do not change. The overall APL decreases from* 2.875 *to* 2.625. *(End of Example)*

Example 4.2 *Fig. 5(a) shows the BDD with the variable order* (x_2,x_3,x_1) *for logic function* $f = x_1(x_2 \vee x_3)$. *Assume that*

$$\begin{aligned} P(x_1 = 0) &= 0.6, \quad P(x_1 = 1) = 0.4, \\ P(x_2 = 0) &= 0.3, \quad P(x_2 = 1) = 0.7, \\ P(x_3 = 0) &= 0.8, \quad P(x_3 = 1) = 0.2. \end{aligned}$$

The APL of the BDD in Fig. 5(a) is 2.06. *For the swap of variables* x_3 *and* x_1, *case (d) applies to node* v_2 *and case (f) applies to node* v_3. *Performing this swap yields the BDD shown in Fig. 5(b). It changes the node traversing probabilities of* v_3 *and* v_4 *(a new node). Before the swap, the edge traversing probability of edge from* v_2 *to* v_3, 0.06, *is subtracted from the APL and from the node traversing probabilities of* v_3. *After the swap, the edge traversing probability of edge from* v_2 *to* v_4, 0.12, *is added to the APL and to* v_4. *The overall APL increases from* 2.06 *to* 2.12. *(End of Example)*

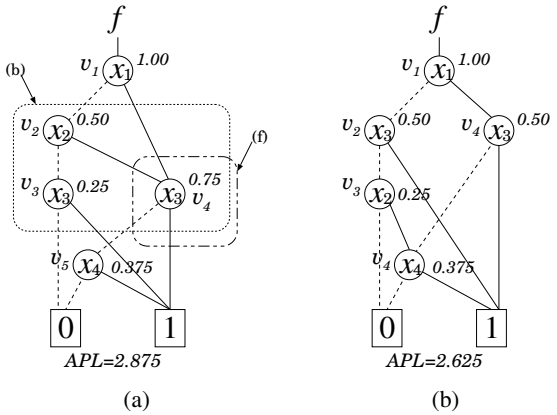


Figure 4: Example of the update of the APL

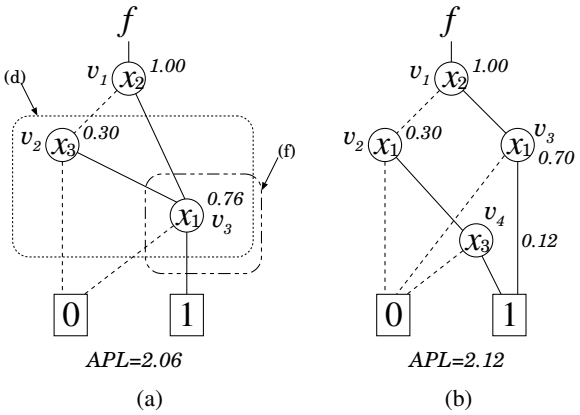


Figure 5: Another example of the update of the APL

4.2 Symmetric Variables

Definition 4.1 *A logic function $f(x_1,x_2,\ldots,x_i,\ldots,x_j,\ldots,x_n)$ is* **symmetric with respect to** x_i **and** x_j *if the interchange of x_i and x_j does not change f. x_i and x_j are called* **symmetric variables**.

In a DD, swapping symmetric variables x_i and x_j does not change the graph structure.

Definition 4.2 *Let π_1 and π_2 be permutations of the variables. If the positions of variables in π_1 are the same as in π_2 except for symmetric variables, π_1 and π_2 are called* **symmetric orders**.

Since symmetric orders produce DDs with the same graph structure, the DDs have the same APL when $P(x_i = 0) = P(x_j = 0)$, $P(x_i = 1) = P(x_j = 1)$, ..., and $P(x_i = r-1) = P(x_j = r-1)$ for symmetric variables x_i and x_j. Therefore, in such a case, detection of symmetric orders can reduce the computation time for an APL minimization algorithm.

Example 4.3 *Consider the logic function $f = x_1x_4 \vee x_2x_4 \vee x_3$ (Fig. 4). Let variable orders π_1 and π_2 be (x_1,x_2,x_3,x_4) and (x_2,x_1,x_3,x_4), respectively. Since x_1 and x_2 are symmetric variables, π_1 and π_2 are symmetric orders. The BDDs for the two orders are the same except the labels x_1 and x_2 are interchanged, and have the same APL and the same number of nodes. (End of Example)*

```
 1:  minimize_APL (DD, input variables X, # inputs n) {
 2:    X_sub = φ ;
 3:    cost[X_sub] = 0 ;
 4:    order[X_sub] = φ ;
 5:    S_next = {X_sub} ;
 6:    min_apl = APL for initial DD ;
 7:    for (level = 1; level ≤ n; level++) {
 8:      S_cur = S_next ;
 9:      S_next = φ ;
10:      for (each X_sub ∈ S_cur) {
11:        ordering(DD, order[X_sub], level - 1) ;
12:        for (each x_i ∈ {X \ X_sub}) {
13:          X'_sub = X_sub ∪ {x_i} ;
14:          Move x_i to level ;
15:          symmetry_check(DD, level) ;
16:          if (order[X'_sub] and current order are symmetric && all P(x = c)s are same)
17:            continue ;
18:          Update min_apl ;
19:          if (lower_bound(level) > min_apl)
20:            continue ;
21:          new_cost = cost[X_sub] + NTP(level) ;
22:          if (new_cost < cost[X'_sub]) {
23:            cost[X'_sub] = new_cost ;
24:            order[X'_sub] = current order ;
25:            if (X'_sub ∉ S_next)
26:              S_next = S_next ∪ {X'_sub} ;
27:          }
28:        }
29:      }
30:    }
31:    ordering(DD, order[X], n) ;
32:  }
```

Figure 6: Exact APL minimization algorithm.

4.3 Exact Minimization Algorithm

Fig. 6 shows a pseudo-code to solve Problem 4.1. This algorithm finds an optimum solution using a branch-and-bound method, similar to the top-down algorithm (*JANUS*) in [9]. *JANUS* [9] uses the number of nodes in a BDD as the cost function, while our algorithm uses the APL of a DD (BDD or MDD) as the cost function. By using the node traversing probability (NTP), the changes in APL can be calculated at each node locally. This locality of computation allows a top-down algorithm. To our knowledge, this is the first time an APL minimization algorithm based on branch-and-bound has been proposed. This algorithm finds an optimum variable order much faster than the exhaustive search method, which enumerates all possible variable orders. In lines 11 and 31 of Fig. 6, procedure *ordering* changes the variable order of the DD into the given order from the top to the specified level. For example, let the current variable order be $(x_1, x_2, x_3, x_4, x_5)$. We seek the order (x_5, x_4) at level two. That is, we seek $(x_5, x_4, *, *, *)$, where "$*, *, *$" represents x_1, x_2, and x_3 in some order. Then, procedure *ordering(DD, (x_5,x_4), 2)* obtains the order $(x_5, x_4, x_1, x_2, x_3)$ in 7 swaps from the order $(x_1, x_2, x_3, x_4, x_5)$. Procedure *symmetry_check* in line 15 checks symmetry of adjacent variables [22]. When the variable order of X'_{sub}, which has already been stored in array "order[X'_{sub}]" as a candidate, and the current variable order of the DD are symmetric, and all $P(x = c)$s are same for the symmetric variables, the current order is excluded from the set of candidates. In line 19, Theorem 3.2 is used to eliminate the unneeded variable exchanges to reduce computation time. In line 21, $NTP(level)$ denotes the sum of the node traversing probabilities of the nodes on the given level (*level*). The initial values of array *cost*

```
 1:  sifting_APL (DD, #rounds of sifting R) {
 2:    cost = APL for initial DD ;
 3:    for (r = 0; r < R; r++) {
 4:      for (each x_i ∈ X) {
 5:        start = current position of x_i ;
 6:        best_p = start ;
 7:        for (each position p from start to the closer extreme) {
 8:          Move x_i to p ;
 9:          Update U (or L) ;
10:          if (cost ≤ U (or L))
11:            break ;
12:          if (APL < cost) {
13:            cost = APL ;
14:            best_p = p ;
15:          }
16:        }
17:        for (each position p to the other extreme) {
18:          Move x_i to p ;
19:          Update U (or L) ;
20:          if (cost ≤ U (or L))
21:            break ;
22:          if (APL < cost) {
23:            cost = APL ;
24:            best_p = p ;
25:          }
26:        }
27:        Move x_i to best_p ;
28:      }
29:    }
30:  }
```

Figure 7: Heuristic APL minimization algorithm.

in Fig. 6 are set to infinity.

4.4 Heuristic Minimization Algorithm

The exact minimization algorithm in Fig. 6 obtains an optimum solution for Problem 4.1. However, when the number of input variables is large, finding the optimum variable order may require much computation time.

In this section, we show a heuristic minimization method using variable sifting [23]. The sifting algorithm repeatedly performs the following basic steps:

1. Change the variable order.
2. Compute a cost.

The proposed sifting algorithm uses APL as the cost function. It was shown in Section 4.1 that the APL can be efficiently updated after the swap of two adjacent variables. As a result, the time needed to compute the cost in our sifting algorithm is comparable to the time needed to update the number of nodes in the classical sifting algorithm, which minimizes the number of nodes. Fig. 7 shows the pseudo-code of the heuristic minimization algorithm. In this algorithm, each variable x_i is sifted across all possible positions to determine its best position. First, x_i is sifted in one direction to the closer extreme (top or bottom). Then, x_i is sifted in the opposite direction to the other extreme. In lines 10 and 20 of Fig. 7, Corollary 3.2 is used to eliminate unneeded sifting of x_i. When variable x_i moves down to the bottom, we use U equal to the sum of the

node traversing probabilities of the nodes above x_i. If $cost \leq U$, sifting of x_i further down to the bottom cannot lead to a smaller APL than $cost$. In such cases, there is no need to continue sifting to the bottom. Similarly, when variable x_i moves up to the top, we use L equal to the sum of the node traversing probabilities of the nodes below x_i. This lower bound for the APL is similar to the one introduced for the number of nodes during the classical sifting [8].

4.5 Initial Ordering of the Binary Variables

The initial ordering of variables influences the effectiveness of the heuristic minimization algorithm described in the previous section. An analysis of variable orders that produces the minimal APL in several known classes of functions [6, 28] leads to a heuristic to find a good initial variable order. In this section, we propose an initial variable order using Walsh spectrum [12] for *binary logic functions*.

The value of a first-order Walsh spectral coefficient expresses the correlation between the variable value with the function value. For n-variable logic function $f(X)$, the first-order Walsh spectral coefficient can be computed as follows [7]:

$$R_i = \frac{|\bar{x}_i \oplus f|}{2^{n-1}} - 1,$$

where $|\bar{x}_i \oplus f|$ denotes the number of assignments of values to the variables X such that the values of x_i and $f(X)$ are equal. The initial variable order is found by placing the variables in descending order of the absolute value of R_i. For variables with identical absolute values of R_i, we arbitrarily choose the order.

All spectral coefficients can be computed by scanning the nodes beginning at the root node and ending on the terminal nodes using a fast algorithm [31]. The first-order coefficients can be computed by a simplified version of the general algorithm.

Example 4.4 *Consider the logic function $f = x_1x_4 \vee x_2x_4 \vee x_3$ in Example 4.1. For each binary variable x_i, the value of $|\bar{x}_i \oplus f|$ is given by:*

$$|\bar{x}_1 \oplus f| = 9, \qquad |\bar{x}_2 \oplus f| = 9, \qquad |\bar{x}_3 \oplus f| = 13, \qquad |\bar{x}_4 \oplus f| = 11.$$

The value of each R_i corresponding to x_i is as follows:

$$R_1 = \frac{1}{8}, \qquad R_2 = \frac{1}{8}, \qquad R_3 = \frac{5}{8}, \qquad R_4 = \frac{3}{8}.$$

Therefore, we have an initial variable order x_3, x_4, x_1, x_2, and APL $= 1.875$. This is the minimum APL for f. (End of Example)

5 PAIRED ORDERING OF BINARY VARIABLES

Unfortunately, there are no standard benchmark functions for multi-valued logic. Thus, 4-valued input 2-valued output functions obtained by pairing binary variables of 2-valued benchmark functions are often used for experiments for the multi-valued case [11, 18, 24]. Especially, [11, 24] show that 4-valued MDDs can represent binary logic functions more compactly than BDDs, by considering paired orderings of binary variables. 4-valued MDDs can be implemented efficiently using LUT-based FPGAs. In Section 6.2, we also use the 4-valued input 2-valued output functions for experiments of the multi-valued case, and show that 4-valued MDDs can reduce the APL, as well as the number of nodes efficiently. To do this, in this section, we define a paired ordering of binary variables.

Definition 5.1 *Let $f(X)$ be a 2-valued logic function, where $X = (x_1, x_2, \ldots, x_n)$ is an ordered set of binary variables. Let $\{X\}$ denote the unordered set of variables in X. Let $X_i \subseteq X$. If $\{X\} = \{X_1\} \cup \{X_2\} \cup \ldots \cup \{X_u\}$, $\{X_i\} \cap \{X_j\} = \phi$ $(i \neq j)$, and $|X_i| = 2$, then $(X_1, X_2, \ldots, X_u)$ is a* **paired ordering** *of binary variables X, and each X_i can be represented as a 4-valued variable. And then, a 2-valued logic function $f(X)$ can be represented by the mapping $f(X_1, X_2, \ldots, X_u)$: $\{0,1,2,3\}^u \rightarrow \{0,1\}$.*

For n-variable functions, if $n < 2u$ (i.e. n is an odd number), we use an additional redundant binary variable called a **dummy variable**. The set of binary variables with the dummy variable if it exists, is denoted by $\{X'\} = \{x_1, x_2, \ldots, x_n, x_{n+1}\}$, where $|X'| = n + 1 = 2u$. Note that f is independent of x_{n+1}.

Theorem 5.1 *The number of different paired orderings of binary variables* $X = (x_1, x_2, \ldots, x_n)$ *to consider is*

$$\frac{n!}{2^u},$$

where u denotes the number of 4*-valued variables obtained by the pairing, and is given by*

$$u = \frac{n}{2}.$$

Note that we assume that $n = 2u$.[2]

Proof. See Appendix.

In [11, 24], heuristic paired ordering algorithms for node minimization have been proposed. However, in this paper, we consider the paired ordering algorithms for APL minimization. We formulate the APL minimization problem considering the paired orderings of binary variables as follows:

Problem 5.1 *Given a binary logic function* $f(X)$*, find a paired ordering of binary variables X that produces an MDD with the minimum APL.*

5.1 Exact Paired Ordering Algorithm

Fig. 8 shows a pseudo-code to solve Problem 5.1. This algorithm finds an optimum solution for Problem 5.1 using branch-and-bound, similar to the algorithm in Fig. 6. In lines 6 and 38 of Fig. 8, procedure *pairing* produces an MDD for 4-valued input 2-valued output function by making pairs of binary variables from the given variable order. In line 7, *min_apl* is set to the APL of MDD obtained by making pairs of binary variables from the variable order of given BDD. In line 26, $pairing\_NTP(level)$ denotes the sum of the node traversing probabilities of the MDD nodes obtained by pairing binary variables in *level* and *level* + 1.

5.2 Heuristic Paired Ordering Algorithm

In this section, we show a heuristic paired ordering algorithm. The heuristic paired ordering algorithm, called *pair-sifting algorithm*, consists of the following four basic steps:

1. Apply the sifting algorithm for APL minimization presented in Section 4.4 to the BDD of the given binary logic function.
2. Make pairs of binary variables from the variable order obtained by the sifting algorithm.
3. Construct an MDD for the 4-valued input 2-valued output function.
4. Apply the sifting algorithm to the MDD.

This strategy is similar to one used in [24], which minimizes the number of nodes in an MDD.

6 EXPERIMENTAL RESULTS

Experiments using MCNC benchmarks were conducted in the following environment:

- CPU: Pentium4 Xeon 2.8GHz
- L1 Cache: 32KB
- L2 Cache: 512KB
- Main Memory: 4GB
- Operating System: redhat (Linux 7.3)
- C-Compiler: gcc -O2

In this section, we assume that $P(x_i = 0) = P(x_i = 1) = \ldots = P(x_i = r-1) = \frac{1}{r}$ for r-valued input functions.

[2] When n is an odd number, we use a dummy variable.

```
 1:  pairing_APL (BDD, binary variables X, # inputs n) {
 2:    X_sub = φ ;
 3:    cost[X_sub] = 0 ;
 4:    order[X_sub] = φ ;
 5:    S_next = {X_sub} ;
 6:    pairing(X, order[X]) ;
 7:    min_apl = APL for initial MDD obtained by pairing ;
 8:    for (level = 1; level ≤ n; level+= 2) {
 9:      S_cur = S_next ;
10:      S_next = φ ;
11:      for (each X_sub ∈ S_cur) {
12:        ordering(BDD, order[X_sub], level - 1) ;
13:        for (each x_i ∈ {X \ X_sub}) {
14:          X'_sub = X_sub ∪ {x_i} ;
15:          Move x_i to level ;
16:          symmetry_check(BDD, level) ;
17:          for (each x_j ∈ {X \ X'_sub}) {
18:            X''_sub = X'_sub ∪ {x_j} ;
19:            Move x_j to level + 1 ;
20:            symmetry_check(BDD, level + 1) ;
21:            if (order[X''_sub] and current order are symmetric && all P(x = c)s are same)
22:              continue ;
23:            Update min_apl ;
24:            if (lower_bound(level + 1) > min_apl)
25:              continue ;
26:            new_cost = cost[X_sub] + pairing_NTP(level) ;
27:            if (new_cost < cost[X''_sub]) {
28:              cost[X''_sub] = new_cost ;
29:              order[X''_sub] = current order ;
30:              if (X''_sub ∉ S_next)
31:                S_next = S_next ∪ {X''_sub} ;
32:            }
33:          }
34:        }
35:      }
36:    }
37:    ordering(BDD, order[X], n) ;
38:    pairing(X, order[X]) ;
39:  }
```

Figure 8: Exact paired ordering algorithm for APL minimization.

6.1 Binary Case

Table 1 compares the number of nodes and APL of BDDs optimized using four different methods: (a) exact minimization of the number of nodes; (b) exact minimization of the APL; (c) the algorithm in [16]; and (d) the heuristic APL minimization algorithm presented in this paper. In the table, *Name* lists the names of benchmark functions. *In* and *Out* lists the numbers of input variables and single-output functions, respectively. Columns *Nodes* contain the number of non-terminal nodes. Columns *Time* contain the CPU time of three algorithms coded by us, in seconds. Unfortunately, the CPU time of the algorithm in [16] is unavailable. Columns "(a) Min_Nodes", "(b) Min_APL", "(c) Liu [16]", and "(d) sifting" show the exact nodes minimization algorithm in [9], the exact APL minimization algorithm in Section 4.3, the heuristic APL minimization in [16], and the heuristic APL minimization in Section 4.4, respectively. Initial variable order for "(d) sifting" was obtained using Walsh spectrum described in Section 4.5. The BDDs in this table use complemented edges. Table 1 includes the same benchmark functions as the experiment in [16] except for incompletely specified functions. We omitted incompletely specified functions because the number of nodes and the APL in BDDs for incompletely specified functions depend on the

Table 1: Minimization of APL for individual BDDs

Name	In	Out	(a) Min_Nodes			(b) Min_APL			(c) Liu [16]		(d) sifting		
			Nodes	APL	Time	Nodes	APL	Time	Nodes	APL	Nodes	APL	Time
5xp1	7	10	66	34.13	0.01	81	31.28	0.01	91	31.31	79	31.28	0.01
alu4	14	8	448	41.75	22.76	547	39.69	28.71	899	47.54	516	39.97	0.01
b12	15	9	64	23.86	0.03	68	21.84	0.01	81	22.22	71	21.88	0.01
con1	7	2	14	6.06	0.01	16	5.94	0.01	16	6.06	16	5.94	0.01
cordic	23	2	73	13.74	416.57	89	9.43	1006.08	259	11.82	88	9.47	0.01
sao2	10	4	99	10.90	0.26	116	10.59	0.06	128	10.71	121	10.59	0.01
vg2	25	8	202	31.00	6431.83	222	29.91	376.78	230	30.37	204	30.16	0.01
misex1	8	7	54	23.22	0.01	57	21.97	0.02	68	22.16	64	21.97	0.01
cm150a	21	1	32	3.50	1106.23	32	3.50	1510.58	33	3.50	32	3.50	0.01
cm151a	12	2	32	6.00	0.38	32	6.00	0.28	36	6.50	32	6.00	0.01
cm162a	14	5	41	11.76	0.06	52	11.70	0.05	59	11.70	48	11.71	0.01
cm163a	16	5	35	11.70	0.01	38	11.70	0.01	42	11.70	36	11.70	0.01
cm85a	11	3	38	7.72	0.05	38	7.72	0.01	47	8.28	38	7.72	0.01
mux	21	1	32	3.50	1098.72	32	3.50	1410.57	33	3.50	32	3.50	0.01
z4ml	7	4	28	18.25	0.01	30	16.38	0.02	32	17.13	28	16.38	0.01
f51m	8	8	51	28.08	0.01	65	27.33	0.02	76	27.45	64	27.45	0.01
pcle	19	9	79	22.50	0.11	84	22.50	0.03	89	22.50	79	22.50	0.01
Average of ratios			1.00	1.00	1.00	1.12	0.95	0.93	1.40	0.99	1.10	0.95	0.40

Table 2: Minimization of APL for shared BDDs for larger functions

Name	In	Out	classical sifting		Coef.	Without Walsh spectrum			With Walsh spectrum		
			Nodes	APL	Time	Nodes	APL	Time	Nodes	APL	Time
C432	36	7	1063	86.58	0.01	1081	86.24	0.15	1899	82.09	0.83
C499	41	32	25873	782.66	0.02	32105	641.16	7.12	32105	641.16	7.11
C880	60	26	4122	140.42	0.01	41701	123.85	4.48	91767	122.22	52.12
C1908	33	25	5532	254.65	0.01	16634	179.20	0.96	13868	171.96	2.73
C2670	233	140	1882	303.34	0.05	2755	278.17	1.30	*	*	*
C3540	50	22	24231	209.15	0.10	25162	208.44	7.44	56898	212.73	75.21
C5315	178	123	1728	460.78	0.05	1820	446.26	0.26	*	*	*
C7552	207	108	2212	485.03	0.05	2207	471.54	0.87	*	*	*
apex3	54	50	931	188.58	0.01	900	158.82	0.04	905	158.73	0.03
apex7	49	37	242	113.88	0.01	277	82.44	0.01	280	82.45	0.02
b9	41	21	108	61.16	0.01	131	55.25	0.01	129	55.39	0.01
dalu	75	16	688	102.67	0.01	990	78.81	0.08	1069	78.81	35.31
des	256	245	3297	1209.50	0.18	3343	1081.13	0.47	3886	1077.63	2.15
duke2	22	29	360	87.89	0.01	386	77.52	0.01	392	77.52	0.02
e64	65	65	128	128.00	0.01	128	128.00	0.01	573	128.00	0.05
ex4	128	28	497	51.38	0.01	629	47.26	0.02	630	47.26	0.03
frg2	143	139	1379	607.00	0.04	1580	322.89	0.15	2189	321.75	0.23
k2	45	45	1257	181.80	0.01	1426	177.52	0.07	1418	177.50	0.10
rot	135	107	7891	446.47	0.05	16164	312.08	5.61	18503	308.68	30.34
Average			1.00	1.00	0.03	1.87	0.85	1.53	3.01	0.84	12.89

* Memory overflow precluded computation of these values.

assignment of values to don't cares, as well as the variable order. To make our results compatible with the results in [16], we optimized each output of the multiple-output benchmark functions independently, and obtained the sum of the values over all outputs. Thus, the number of nodes and APL in Table 1 are different from those of the SBDD. Two rounds of sifting are performed in all experiments. The row labeled *Average of ratios* represents the normalized averages for *Nodes*, *APL*, and *Time* assuming the values of "(a) Min_Nodes" to be 1.00. The columns "(b) Min_APL", "(c) Liu [16]", and "(d) sifting" of this row contains the relative values to the results of "(a) Min_Nodes".

The heuristic method in [16] obtained BDDs with the exact minimum APLs in 5 out of 17 benchmark functions. However, for *alu4*, *cm151a*, and *cm85a*, the algorithm in [16] obtained BDDs with much larger APLs than the exact minimum APLs. On the other hand, our heuristic method in Section 4.4 obtained BDDs with the exact minimum APLs in 11 out of 17 benchmark functions. For five of the remaining functions, the APLs in the column labeled "(d) sifting" are smaller than or equal to the APLs in "(c) Liu [16]". For *cm162a*, our sifting algorithm obtained BDDs with slightly larger APLs than the exact minimum APLs.

An exhaustive search algorithm finds the minimum APLs for the functions with up to 14 inputs within a reasonable computation time. Meanwhile, our exact minimization algorithm in Section 4.3 found the minimum APL for functions with 25 inputs (*vg2*) within a reasonable computation time.

Table 2 shows the results for larger MCNC benchmarks and the effectiveness of the initial variable order using the Walsh spectrum. In this table, we used SBDDs with complemented edges for multiple-output functions. In Table 2, the column "classical sifting" shows the number of nodes and APL for BDDs obtained by the sifting algorithm [23] which minimizes the number of nodes in BDD. The column "Without Walsh spectrum" shows the results of our sifting algorithm, which minimizes the APL, where the initial variable orders are the variable orders of BDDs obtained by "classical sifting". And, the column "With Walsh spectrum" shows the results of our sifting algorithm, where the initial variable orders were obtained using Walsh spectrum shown in Section 4.5. The column "Coef. Time" denotes the CPU time needed to calculate the values of first-order Walsh spectral coefficients R_i, in seconds. Unfortunately, for *C2670*, *C5315*, and *C7552*, BDDs with the initial variable orders could not be constructed due to memory overflow. The row labeled *Average* represents average of *Time* and normalized averages of *Nodes* and *APL* assuming the values of "classical sifting" to be 1.00. The columns "Without Walsh spectrum" and "With Walsh spectrum" show the relative values to the results of "classical sifting".

For some benchmark functions, for example, *C1908*, *frg2*, and *rot*, the APLs are reduced drastically. For *C7552*, the number of nodes is reduced as a byproduct of the APL minimization. However, for most functions, the number of nodes is increased by the APL minimization. The comparison of "Without Walsh spectrum" and "With Walsh spectrum" shows the effectiveness of the initial variable order using Walsh spectrum. For 8 out of 19 benchmark functions, the APLs in the column "With Walsh spectrum" are smaller than the APLs in "Without Walsh spectrum". The computation time to calculate the values of R_i is short.

However, for most functions, the computation times of sifting for "With Walsh spectrum" are significantly longer than that for "Without Walsh spectrum" because the number of nodes in BDD with initial variable order computed using Walsh spectrum is large. When the number of nodes in the BDD is large, swapping one pair of adjacent variables takes a longer time because the time needed for the swap is roughly proportional to the number of nodes present on the given levels in the BDD.

Tables 1 and 2 show that the proposed heuristic minimization minimizes the APL in short computation time. For small benchmark functions in Table 1, the heuristic minimization could obtain BDDs with near-minimum APLs. For large benchmark functions in Table 2, the heuristic algorithm reduces APLs to 84% on the average.

6.2 Multi-Valued Case

There are no standard benchmark functions for multi-valued logic. Thus, by pairing binary variables of 2-valued benchmark functions, we obtained 4-valued input 2-valued output functions. Table 3 compares the number of nodes and the APLs of the BDDs and the MDDs optimized using four algorithms: (a) exact minimization of the APL for BDDs; (b) exact minimization of the number of nodes for MDDs; (c) exact minimization of the APL for MDDs; and (d) the heuristic APL minimization algorithm for MDDs. Columns "BDD", "Min_Nodes", "Min_APL", and "pair-sifting" denote the exact APL minimization algorithm in Section 4.3, the exact paired ordering algorithm for node minimization, the exact paired ordering algorithm for APL minimization in Section 5.1, and the pair-sifting algorithm in Section 5.2, respectively. The symbol '*' in this table denotes that the results could not be obtained because of memory overflow. The bottom row labeled *Average* represents normalized averages of *Nodes*, *APL*, and *Time* for all functions except for 4 functions (*cordic*, *cm150a*, *mux*, *pcle*), where the values of "BDD" are set to 1.00. The SBDDs and SMDDs in this table do not use complemented edges. Note that the values (Nodes, APL, Time) of BDDs in this table are different from the values in Table 1, because in this table, SBDDs

Table 3: Comparison of APLs for SBDDs and SMDDs

	BDD			MDD								
				Min_Nodes			Min_APL			pair-sifting		
Name	Nodes	APL	Time	Nodes	APL	Time	Nodes	APL	Time	Nodes	APL	Time
5xp1	68	32.00	0.01	37	20.69	0.01	37	20.69	0.02	37	20.69	0.01
alu4	462	40.70	47.96	247	27.49	77.46	290	23.26	101.64	393	27.00	0.01
b12	66	22.77	2.26	39	22.84	387.21	50	16.83	5.11	55	19.00	0.01
con1	18	6.31	0.01	11	4.63	0.01	11	4.06	0.01	13	5.69	0.01
cordic	94	9.46	3996	*	*	*	43	5.21	3587	70	5.81	0.01
sao2	102	10.64	0.12	46	9.98	0.21	58	6.57	0.07	55	6.79	0.01
misex1	46	22.84	0.02	22	15.50	0.01	28	12.75	0.01	30	15.41	0.01
cm150a	32	3.50	5972	*	*	*	13	2.25	19509	26	2.47	0.01
cm151a	32	6.00	0.47	16	5.50	0.13	22	4.00	0.80	22	4.00	0.01
cm162a	38	11.70	0.86	19	8.73	0.55	27	8.44	0.56	23	8.66	0.01
cm163a	40	11.70	1.98	19	12.25	19.60	22	8.16	1.11	19	8.16	0.01
cm85a	37	7.72	0.08	15	6.28	0.08	16	5.38	0.16	25	5.84	0.01
mux	32	3.50	6334	*	*	*	13	2.25	13811	26	2.47	0.01
z4ml	26	16.38	0.01	10	9.13	0.01	10	9.13	0.01	10	9.13	0.01
f51m	76	28.02	0.03	39	18.81	0.02	41	17.38	0.03	41	18.81	0.01
pcle	83	22.50	34.46	*	*	*	50	16.50	29.56	48	20.92	0.01
Average	1.00	1.00	1.00	0.50	0.79	15.89	0.59	0.64	1.28	0.63	0.70	0.34

* Memory overflow precluded computation of these values.

Table 4: Minimization of APL for SMDDs for larger functions

Name	Node pair-sifting [24]			APL pair-sifting		
	Nodes	APL	Time	Nodes	APL	Time
C432	617	59.84	0.03	721	58.75	0.15
C499	13541	407.23	1.52	16397	339.73	9.23
C880	3025	118.99	0.30	34730	107.89	9.35
C1908	4390	167.42	0.55	15287	124.36	1.49
C2670	2336	276.19	0.31	3945	260.65	1.99
C3540	22519	155.06	7.33	24241	157.57	19.16
C5315	1947	398.53	0.23	2258	393.20	0.49
C7552	2292	420.69	0.45	2236	431.77	1.86
apex3	628	143.66	0.03	694	96.72	0.06
apex7	200	99.82	0.02	257	73.29	0.01
b9	126	55.90	0.01	173	51.66	0.02
dalu	523	70.55	0.03	644	42.93	0.13
des	2685	934.38	0.55	2994	911.44	1.04
duke2	272	65.99	0.02	288	51.50	0.01
e64	96	96.67	0.01	993	86.44	0.03
ex4	420	46.00	0.02	482	39.45	0.05
frg2	1179	544.34	0.10	1457	260.11	0.20
k2	1055	168.17	0.05	912	108.42	0.11
rot	5615	393.57	1.29	14898	284.22	9.52
Average	1.00	1.00	1.00	2.40	0.83	4.66

without complemented edges are used.

The pair-sifting algorithm obtained MDDs with the exact minimum APL for 4 functions. On the average, the pair-sifting algorithm reduced the APL to 70% of "BDD". For *con1*, the pair-sifting algorithm obtained larger APL than that of "Min_Nodes" due to heuristic pairing algorithm. However, this algorithm can obtain a smaller APL and fewer nodes than those of the corresponding BDD. Although the exact paired ordering algorithms for nodes and APL can reduce both nodes and APL drastically, they are time-consuming. On the other hand, the pair-sifting algorithm quickly reduces both Nodes and APL.

Table 4 shows the results for larger MCNC benchmark functions. Similarly, we obtained 4-valued input 2-valued output functions by pairing binary variables. Column "Node pair-sifting [24]" denotes the heuristic paired ordering algorithm for the node minimization method proposed in [24]. The number of nodes in an MDD obtained by the paired ordering algorithm for node minimization is smaller than or equal to the corresponding BDD [24]. However, since the MDDs in this table do not use the complemented edges, some MDDs are larger than the BDDs with complemented edges in Table 2. The bottom row labeled *Average* represents normalized averages of *Nodes*, *APL*, and *Time* assuming the values of "Node pair-sifting [24]" to be 1.00.

For these benchmark functions, the pair-sifting algorithm reduced the APL to 83% of "Node pair-sifting [24]", on average. Especially, for *frg2*, the APL was reduced to 48% of "Node pair-sifting [24]". The pair-sifting algorithm cannot always find an MDD with the minimum APL, because it is a heuristic algorithm. For *C3540* and *C7552*, the APLs are slightly larger than that in "Node pair-sifting [24]". However, Tables 2 and 4 show that the pair-sifting algorithm can find an MDD with smaller APL than APL of corresponding BDD.

7 CONCLUSION AND COMMENTS

We have proposed an exact and a heuristic algorithm for the minimization of the APL in BDDs and MDDs. The experimental results using MCNC benchmark functions show that: 1) The exact minimization algorithm finds BDDs with the minimum APL for the function with up to 25 input variables within a reasonable computation time. 2) Using the node and edge traversing probabilities to compute and update the APLs after the swap of two adjacent variables, the proposed sifting algorithm can heuristically minimize the APLs as fast as classical sifting, which minimizes the number of nodes. 3) Using an initial variable order computed using Walsh spectral coefficients increases the quality of the results of APL minimization algorithms. However, in some cases the initial variable order leads to BDDs with a large number of nodes, which slows down

APL minimization. 4) MDDs produced by pairing binary variables have smaller APL and fewer nodes than corresponding BDDs.

ACKNOWLEDGMENTS

This research is partly supported by the Grant in Aid for Scientific Research of the Japan Society for the Promotion of Science (JSPS), and the funds from Ministry of Education, Culture, Sports, Science, and Technology (MEXT) via Kitakyushu innovative cluster project. We also thank the reviewers for constructive comments.

REFERENCES

[1] Ashar, P. and Malik, S. (1995). Fast functional simulation using branching programs, *ICCAD'95*, 408–412.

[2] Balarin, F., Chiodo, M., Giusto, P., Hsieh, H., Jurecska, A., Lavagno, L., Sangiovanni-Vincentelli, A., Sentovich, E. M., and Suzuki, K. (1999). Synthesis of software programs for embedded control applications, *IEEE Trans. CAD*, 18(6), 834–849.

[3] Bertacco, V., Minato, S., Verplaetse, P., Benini, L., and De Micheli, G. (1997). Decision diagrams and pass transistor logic synthesis, *International Workshop on Logic and Synthesis*, Lake Tahoe, s. 3-3.

[4] Brglez, F. and Fujiwara, H. (1985). Neutral netlist of ten combinational benchmark circuits and a target translator in FORTRAN, *Special session on ATPG and fault simulation, Proc. IEEE Int. Symp. Circuits and Systems*, 663–698.

[5] Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation, *IEEE Trans. Comput.*, C-35(8), 677–691.

[6] Butler, J. T. and Sasao, T. (2003). On the average path length in decision diagrams of multiple-valued functions, *33rd International Symposium on Multiple-Valued Logic*, Tokyo, Japan, 383–390.

[7] Dertouzos, M. L. (1965). Threshold Logic: A Synthesis Approach, Mass. Inst. Tech., Cambridge, Res. Monograph 32. Cambridge, Mass.: M. I. Press.

[8] Drechsler, R., Günther, W., and Somenzi, F. (2001). Using lower bounds during dynamic BDD minimization, *IEEE Trans. CAD*, 20(1), 51–57.

[9] Ebendt, R., Günther, W., and Drechsler, R. (2003). Combination of lower bounds in exact BDD minimization, *Design Automation and Test in Europe conference and exhibition (DATE'03)*, Munich, Germany, 758–763.

[10] Fujita, M., Matsunaga, Y., and Kakuda, T. (1991). On variable ordering of binary decision diagrams for the application of multi-level logic synthesis, *EDAC*, 50–54.

[11] Hafiz Md. Hasan Babu and Sasao, T. (2000). Heuristics to minimize multiple-valued decision diagrams, *IEICE Trans. on Fundamentals*, E83-A(12), 2498–2504.

[12] Hurst, S. L., Miller, D. M., and Muzio, J. C. (1985). *Spectral techniques in digital logic*, Academic Press., London.

[13] Iguchi, Y., Sasao, T., Matsuura, M., and Iseno, A. (2000). A hardware simulation engine based on decision diagrams, *Asia and South Pacific Design Automation Conference (ASP-DAC'2000)*, Yokohama, Japan, 73–76.

[14] Jiang, Y. and Brayton, B. K. (2002). Software synthesis from synchronous specifications using logic simulation techniques, *Design Automation Conference*, New Orleans, LA, U.S.A, 319–324.

[15] Kam, T., Villa, T., Brayton, R. K., and Sangiovanni-Vincentelli, A. L. (1998). Multi-valued decision diagrams: Theory and Applications, *Multiple-Valued Logic: An International Journal*, 4(1-2), 9–62.

[16] Liu, Y. Y., Wang, K. H., Hwang, T. T., and Liu, C. L. (2001). Binary decision diagrams with minimum expected path length, *Proc. DATE 01*, 708–712.

[17] McGeer, P. C., McMillan, K. L., Saldanha, A., Sangiovanni-Vincentelli, A. L., and Scaglia, P. (1995). Fast discrete function evaluation using decision diagrams, *ICCAD'95*, 402–407.

[18] Miller, D. M. and Drechsler, R. (1998). Implementing a multiple-valued decision diagram package, *Proc. 28th Int. Symp. on Multiple-Valued Logic*, 52–57.

[19] Minato, S., Ishiura, N., and Yajima, S. (1990). Shared binary decision diagram with attributed edges for efficient Boolean function manipulation, *Proc. 27th ACM/IEEE Design Automation Conf.*, 52–57.

[20] Nagayama, S., Mishchenko, A., Sasao, T., and Butler, J. T. (2003). Minimization of average path length in BDDs by variable reordering, *International Workshop on Logic and Synthesis*, Loguna Beach, California, U.S.A., 207–213.

[21] Nagayama, S. and Sasao, T. (2003). Compact representations of logic functions using heterogeneous MDDs, *IEICE Trans. on Fundamentals*, E86-A(12), 3168–3175.

[22] Panda, S., Somenzi, F., and Plessier, B. F. (1994). Symmetry detection and dynamic variable ordering of decision diagrams, *ICCAD*, San Jose, CA, 628–631.

[23] Rudell, R. (1993). Dynamic variable ordering for ordered binary decision diagrams, *ICCAD'93*, 42–47.

[24] Sasao, T. and Butler, J. T. (1996). A method to represent multiple-output switching functions by using multi-valued decision diagrams, *26th International Symposium on Multiple-Valued Logic*, Santiago de Compostela, Spain, 248–254.

[25] Sasao, T. and Fujita, M. (ed.) (1996). *Representations of Discrete Functions*, Kluwer Academic Publishers.

[26] Sasao, T. (1999). *Switching Theory for Logic Synthesis*, Kluwer Academic Publishers.

[27] Sasao, T., Iguchi, Y., and Matsuura, M. (2002). Comparison of decision diagrams for multiple-output logic functions, *International Workshop on Logic and Synthesis*, New Orleans, Louisiana, 379–384.

[28] Sasao, T., Butler, J. T., and Matsuura, M. (2002). Average path length as a paradigm for the fast evaluation of functions represented by binary decision diagrams, *International Symposium on New Paradigm VLSI Computing*, Sendai, Japan, 31–36.

[29] Shelar, R. S. and Sapatnekar, S. S. (2002). Efficient layout synthesis algorithm for pass transistor logic circuits, *Asia and South Pacific Design Automation Conference (ASP-DAC'2002)*, Bangalore, India, 87–92.

[30] Shelar, R. S. and Sapatnekar, S. S. (2002). Efficient layout synthesis algorithm for pass transistor logic circuits, *International Workshop on Logic and Synthesis*, New Orleans, Louisiana, 209–214.

[31] Thornton, M., Miller, D. M., and Drechsler, R. (2001). Transformations amongst the Walsh, Haar, arithmetic and Reed-Muller spectral domains, *Proc. Intl. Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, 215–225.

[32] Yang, C. and Ciesielski, M. (2002). BDS: A BDD-based logic optimization system, *IEEE Trans. CAD*, 21(7), 866–876.

[33] Yang, S. (1991). *Logic synthesis and optimization benchmark user guide version 3.0*, MCNC.

APPENDIX

Proof for Lemma 2.1. We prove only the first statement; the proof for the second statement is similar. Consider a node v. Any path that includes an incoming edge to v includes v. Conversely, any path that includes v includes an incoming edge to v. It follows that any assignment of values to the variables that corresponds to a path through v contributes to the node traversing probability of v an amount that is identical to the amount contributed to the edge traversing probability of an incoming edge to v. It follows that the node traversing probability of v is equal to the sum of edge traversing probabilities of all incoming edges to v. □

Proof for Theorem 2.1. We prove only the first statement; the proof for the second statement is similar. From Definition 2.6, we have

$$ETP(e) = \sum_{p \in SP(e)} PP(p), \tag{1}$$

where $SP(e)$ is a set of paths including the edge e. We prove the following

$$APL = \sum_{i=1}^{N_e} ETP(e_i), \tag{2}$$

where N_e denotes the number of edges in a DD. From (1), (2) can be transformed as follows:

$$\begin{aligned} APL &= \sum_{i=1}^{N_e} ETP(e_i) \\ &= \sum_{i=1}^{N_e} \sum_{p \in SP(e_i)} PP(p) \end{aligned} \tag{3}$$

From Definition 2.4, we have

$$\begin{aligned} APL &= \sum_{i=1}^{N} PP(p_i) \times l_i \\ &= \sum_{i=1}^{N} \sum_{j=1}^{l_i} PP(p_i) \end{aligned} \tag{4}$$

Although (3) and (4) use different computational approaches, they obviously compute the same value. □

Proof for Lemma 3.1. An SMDD for $F = (f_0, f_1, \ldots, f_{m-1})$ is traversed from a root node to a terminal node m times to evaluate multiple-output function F. Since m_U root nodes are located above or in level i, m_U traversals via edges in $Cut(i)$ are performed while evaluating the multiple-output function. Therefore, we have $ETP(Cut(i)) = m_U$. □

Proof for Lemma 3.2. From Lemma 2.1, the following relation holds:

$$ETP(Cut'(i)) = \sum_{v \in V_c} NTP(v),$$

where V_c denotes a set of non-terminal nodes representing the cofactors with respect to X_{upper}. The probability of the occurrence of the cofactor depends only on the function and not the order of X_{upper}. Since $Cut'(i)$ does not include the edges to terminal nodes, the upper bound of m_U on c_i follows from Lemma 3.1. □

Proof for Theorem 3.1. All nodes representing cofactors with respect to the variables in X_{upper} and m_L root nodes are situated below or in level $i+1$. Thus, L includes the node traversing probabilities of these nodes. □

Proof for Theorem 3.2. Let L be the sum of the node traversing probabilities of the non-terminal nodes below or in level $i+1$. From Theorem 2.1, we have

$$APL = U + L.$$

Then, from Theorem 3.1, for any permutation of X_{lower},

$$APL \geq U + ETP(Cut'(i)) + m_L.$$

□

Proof for Theorem 4.1. The variable swap of level i and level $i+1$ does not influence the graph structure except for levels i and $i+1$ because of the locality of the swap operation. Thus, it is clear that U remains unchanged. From Lemma 2.1, L is obtained by the sum of $ETP(Cut'(i+1))$ and $ETP(E_{lower})$, where

$$\begin{aligned} &Cut'(i+1) = \{e \mid e \in Cut(i+1),\ e \text{ is incident to a non-terminal node}\}, \\ &E_{lower} = \{e \mid e \text{ is an edge situated below or in level } i+2\}, \\ &ETP(Cut'(i+1)) = \sum_{e \in Cut'(i+1)} ETP(e), \\ &ETP(E_{lower}) = \sum_{e \in E_{lower}} ETP(e). \end{aligned}$$

By Lemma 3.2, $ETP(Cut'(i+1))$ is an invariant. $ETP(E_{lower})$ remains unchanged because of the invariance of $ETP(Cut'(i+1))$ and the locality of the swap operation. Therefore, L also remains unchanged. □

Proof for Theorem 5.1. The number of different permutations of binary variables X is $n!$. Since from Definition 5.1, the binary variables X are partitioned into the unordered sets $\{X_1\},\{X_2\},\ldots,\{X_u\}$, the order of binary variables in each $\{X_i\}$ is not important. The number of different permutations of two binary variables in each $\{X_i\}$ is 2. Therefore, we have the theorem. □